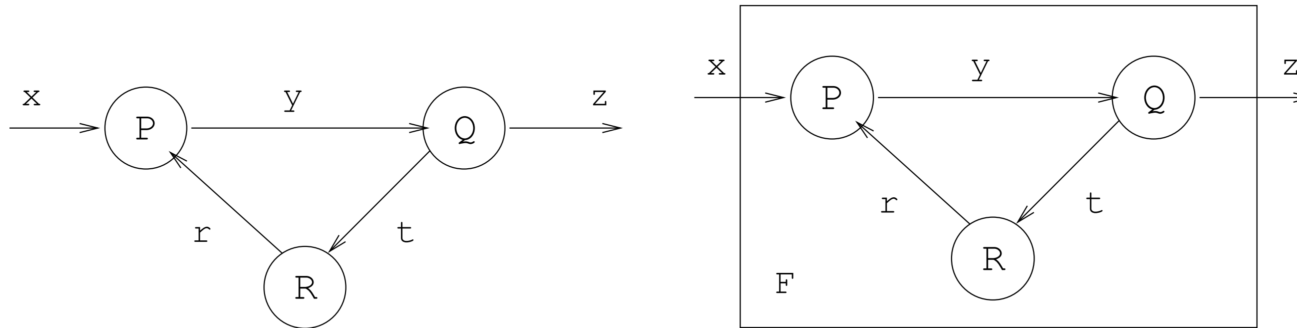
Data-flow Parallelism with FIFOs Communications

Semantics of Block-diagram Languages

Block-diagram languages : A system is defined by a set of operators and oriented wires. E.g. :



are represented by a set of equations (left) and an equation (right) :

$$\begin{aligned} z, t &= Q(y) \\ y &= P(x, r) \\ t &= R(t) \end{aligned} \qquad\qquad z = F(x)$$

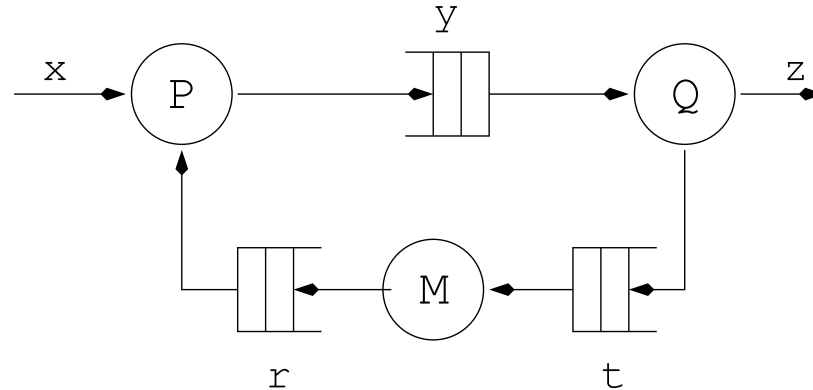
What is the meaning (semantics) of these two sets of equations : Meaning of z, t, y, P, Q, R ? Knowing them, what is the meaning of F ?

Modularity : the two (left and right) sets of equations should define the same relation between x and z, i.e., naming a set of equations should not change the semantics of the system.

Kahn networks

Kahn Networks [Kahn'74, Kahn'75]

In the 70's Kahn showed that the semantics of deterministic parallel processes communicating through (possibly) unbounded buffers is a stream function.



— A set of sequential deterministic processes (i.e., sequential programs) written in an imperative language : `P`, `Q`, `M`,...

— They communicate **asynchronously** via **message passing** into FIFOs (buffers) using two primitives `get/put` with the following assumptions :

  — Read is blocking on the empty FIFO ; sending is non blocking.

  — Channels are supposed reliable (communication delays are bounded).

  — Read (waiting) on a single channel only, i.e., the program :

```
if (a is not empty) or (b is not empty) then ...
```

  is FORBIDDEN

Concretely :

- A buffer channel is defined by two primitives, `get`, to wait (pop) a value from a buffer and `put`, to send (push) a value.
- Parallel composition can either be implemented with regular processes ("fork") or lightweight processes ("threads").
- Historically, Gilles Kahn was interested in the semantics of Unix pipes and Unix processes communicating through FIFOs.

E.g., take OCaml :

```
type 'a buff = { put: unit -> 'a; get: 'a -> unit }
val buffer : unit -> 'a buff
```

`buffer ()` creates a buffer associated to a read and write functions.

Either unbounded size (using lists) or statically bounded size. Add a possible status bit : `IsEmptyBuffer` and `IsFullBuffer`.

Here is a possible implementation of the previous set of processes (`kahn.ml`).

```
(* Process P *)
let p x r y () =
  y.put 0; (* init *)
  let memo = ref 0 in
  while true do
    let v = x.get () in
    let w = r.get () in
    memo := if v then 0 else !memo + w;
    y.put !memo
  done
```

```
(* Process Q *)
let q y t z () =
  while true do
    let v = y.get () in
    t.put v;
    z.put v
  done
```

```
(* Process M *)
let m t r () =
  while true do
    let v = t.get () in
    r.put (v + 1)
  done
```

```
(* Put them in parallel. *)
let main x z () =
  let r = buffer () in let y = buffer () in
  let t = buffer () in
  Thread.create (p x r y) ();
  Thread.create (p y t z) ();
  Thread.create (m t r) ()
```

Question :

- Provide an implementation of the function `buffer` in OCaml (using modules `Thread`, `Mutex`, or `Unix` and `Sys`).

Questions :

- What is the semantics of `p`, `q`, `m` and `main` ?
- What does it change when removing line `(* init *)` ?
- Would you be able to prove that the program `main` is non blocking, i.e, if `x.get ()` never blocks then `z.put ()` never blocks ?
- Is there a statically computable bound for the size of buffers without leading to blocking ?
- Would it be possible to statically schedule this set of processes, that is, to generate an equivalent sequential program ?

These are all undecidable questions in the general case (see [Thomas Park's PhD. thesis, 1995], among others).

Kahn Process Networks and variants have been (and are still) very popular, both practically, and theoretically as it conciliates **parallelism** and **determinacy**.

Semantics

In his seminal paper, Kahn showed that a process with possible internal state can be interpreted as a **continuous** function from **sequences to sequences**.

Values :

- $V^\infty = V^* + V^\omega$, the set of finite and infinite sequences of elements in V.
- The empty sequence is noted ϵ. $x.y$ for the sequence whose first element is $x \in V$ and rest y. If x and y are two sequences, $x \bullet y$ is the concatenation of the two.
- Prefix order $\leq$ between sequences :
  - For all $x, y \in V^\infty$, $x \leq y$ if it exists z st $x \bullet z = y$.
  - Equivalently, for all $x \in V^\infty, \epsilon \leq x$ and for all $v \in V, x, y \in V^\infty$, if $x \leq y$ then $v.x \leq v.y$

In the remaining, I will often write x_i ($i \in \mathbb{N}$) for the i-th element of x.

What is presented now is a particular case of a more general theory of **denotational semantics** studied by Scott and Strachey in the 70's.

Book : "Theories of Programming Languages", John Reynolds, Cambridge Univ. Press, 1998.

Finite and infinite sequences

- $\leq$ is an order (reflexive, transitive, anti-symetric) ;
- ε is the minimum element ;
- Every increasing chain $x_0 \leq x_1 \leq ... \leq x_n...$ has a least upper-bound noted $\cup_{i=0}^{\infty}(x_i)$ which also an element of V^∞.

Monotony : A stream function $f : V^\infty \mapsto V^\infty$ is monotone iff for all x, y in V^∞, $x \leq y$ implies $f(x) \leq f(y)$.

Continuity : $f : V^\infty \mapsto V^\infty$ is continuous iff $f(\cup_{i=0}^{\infty}(x_i)) = \cup_{i=0}^{\infty}(f(x_i))$

Remark : If $f : V^\infty \mapsto V^\infty$ is continuous, it is also monotone.

(Let $x \leq y$. $f(y) = f(x \cup y) = f(x) \cup f(y)$. Thus, $f(x) \leq f(y)$.)

Theorem : if D is a CPO and f, a continuous function from V^∞ to V^∞, then :

$$x = \mathit{fix}(f) = \cup_{i=0}^{\infty}(f^i(\varepsilon))$$

is the least fixed-point of f, that is, $x = f(x)$ and $x \leq x'$ for any x' such that $x' = f(x')$.

(This is a particular case of a more general theorem, called the “Kleene Fix-point theorem”).

Application to data-flow networks : The Kahn Principle

Kahn proposed the following interpretation to channels and processes.

- A channel of type V is interpreted as an element of V^∞.
- A process is a continuous function from the *history* of its input channel $X = x_1, ..., x_n, ...$ to an *history* of its output channel, i.e., and element of $V^\infty \mapsto V^\infty$.

Remark : this generalizes to MIMO (multiple inputs/multiple outputs).

- a process is a set of stream functions. For each node P with n inputs and k outputs, define k continuous functions :

$$f_1 : V_1^\infty \times V_n^\infty \to V_1'^\infty, \ldots, f_k : V_1^\infty \times V_n^\infty \to V_k'^\infty$$

Monotony : A process is a monotone or *causal*, that is, the output can be produced before all inputs are available. Giving more input does not change the already produced outputs.

Continuity : Moreover, it cannot decide to produce an output once it has received an infinite number of inputs.

Examples

Example of continuous functions :

1. Combinatorial functions : there exists a g such that for all x, s :

$$f(\epsilon) = \epsilon \quad f(x.s) = g(x).f(s)$$

2. A (Moore) sequential function : there exists v, h such that for all x, s :

$$f(\epsilon) = v \quad f(x.s) = v.(h\,x\,s)$$

3. A (Mealy) sequential function : there exists g, h such that for all x, s :

$$f(\epsilon) = \epsilon \quad f(x.s) = (g\,x).(h\,x\,s)$$

Example of non monotone/non continuous functions

An example of a non monotone function ($V = \{0, 1\}$) :

$$f(0) = 0 \qquad f(0.1) = 1$$

An example of a monotone, non continuous function ($V = \{0, 1\}$) :

$$f(s) = 0 \text{ if } s \text{ is a finite stream} \qquad f(s) = 01 \text{ if } s \text{ is a infinite stream}$$

Length preserving functions :

- $|x|$ is the length of a sequence x ($|.| : D^\infty \to \mathbb{N} + \{\infty\}$).
- f is length preserving if for all x, $|x| \leq |f(x)|$
- f is length increasing if for all x, $|x| < |f(x)|$

Examples : combinatorial functions, delays are all length preserving. The set of length preserving functions is closed by composition.

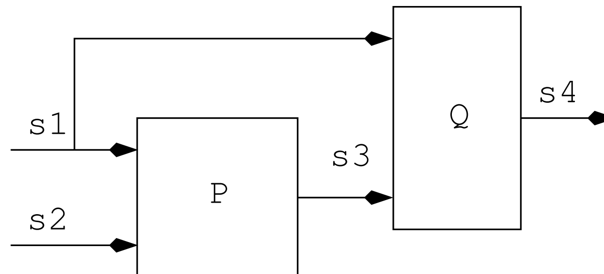
Terminology : A network of operator is said **feed-forward** if its graph is acyclic. A **feed-back** loop stands for a cycle in the graph.

Many systems have feed-back loops : e.g., a heat controller continuously sends and reads values from the environment.

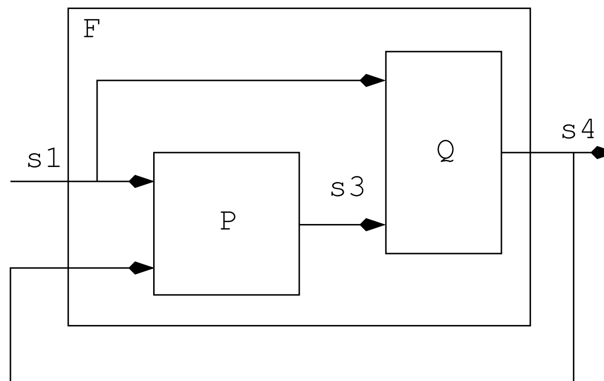
Semantics (composition)

Processes are interpreted by continuous stream functions. Any composition of process also defines a continous stream function.

1. $f(s_1, s_2) = let\ s_3 = p(s_1, s_2)\ in\ q(s_1, s_3)$ is continuous



2. $g(s_1) = let\ s_4 = fix(x \mapsto f(s_1, x))\ in\ s_4$ is continuous.



Feedback loops : the system $\vec{x} = f(y)(\vec{x})$ where $f(y)$ is a continuous function has a least fixed-point $(v_1, ..., v_n) = \mathit{fix}(f(y))$ where :

$$v_i = lim_{k \to \infty}(x_i^k) \qquad \vec{x}^0 = (\epsilon, ..., \epsilon) \qquad \vec{x}^{k+1} = f(y)(\vec{x}^k)$$

Modularity and feedback : The least fix-point of such a system, i.e., $y \mapsto \mathit{fix}(f(y))$, is still a continous function of the parameter y of the system.

Recursion : In the work of Kahn, networks could be dynamic (recursive) and this preserve continuity. E.g., Eratosthene's sieve.

To do (homework) : Read the two papers of Gilles Kahn. Program the sieve of Eratosthene in OCaml using buffer communication.

Application to the introductory example

Build the set of equations from a Kahn Process Network :

- associate a variable x to every channel
- for every node f interpreted as $f_1, ..., f_k$ with inputs $x_1, ..., x_n$ and outputs $y_1, ..., y_k$, define :

$$y_1 = f_1(x_1, ..., x_n), \ldots, y_k = f_k(x_1, ..., x_n)$$

The semantics of the network is the smallest fixed point of the system of equations.

Example : *(cf. above)* the system associated to the introductory example is :

$$\begin{array}{ll} y = P(x) & z = Q_1(y) \\ t = Q_2(y) & r = M(t) \end{array}$$

Definition of P, Q_1, Q_2 and M :

Every channel (e.g., `x`, `r`, `y`, `z`), local ou reference (e.g., `memo`, `v`, etc.) is interpreted as a sequence, i.e., the history of values it contains over the execution.

— $y = P(x, r)$ iff for all $n \in I\!N$:

1. $v_n = x_n$, $w_n = r_n$
2. $memo_{n+1} = \mathit{if}\ v_n\ \mathit{then}\ 0\ \mathit{else}\ memo_n + w_n$ and $memo_0 = 0$
3. $y_n = \mathit{if}\ init\ \mathit{then}\ 0\ \mathit{else}\ memo_{n-1}$
4. $init_0 =$ `true` and $init_n =$ `false`

— $t = Q_1(y)$ if for all $n \in I\!N$, $t_n = y_n$

— $z = Q_2(y)$ if for all $n \in I\!N$, $z_n = y_n$

— $r = M(t)$ if for all $n \in I\!N$, $r_n = t_n$

— $z = main(x)$ if it exists r, y, t such that for all $n \in I\!N$:

$$(y = P(x, r)) \wedge (t = Q_1(y)) \wedge (z = Q_2(y)) \wedge (r = M(t))$$

This is equivalent to $z = P(x, z)$, that is :

$$z_0 = 0 \text{ and } z_{n+1} = \mathit{if}\ x_n\ \mathit{then}\ 0\ \mathit{else}\ z_n + z_n$$

The previous set of processes is thus equivalent to the single process where parallel composition has been eliminated. **Parallelism is compiled**.

```
let main_without_parallel x z () =
  z.put 0;
  let memo = ref 0 in
  while true do
    let v = x.get () in
    memo := if v then 0 else !memo + !memo;
    z.put !memo
  done
```

Instead of programming with assignments (and try to build its denotation), what about directly program with equations ?

The early days of data-flow programming

Instead of writing imperative code with FIFO communication, directly program by writing equations between *histories*. A come back to the early work of Ashcroft and Wadge (70's).

The language LUCID [CACM'77] : Their purpose was to avoid sequential programming and assignment to variables, replacing them by equations between *histories*. E.g., define `main` by a set of equations given in any order : [a]

```
v = x
memo = 0 fby (if v then 0 else memo + memo)
z = memo
```

`fby` stands for "followed by". It returns the value of its first argument then acts as its second. The semantics of this program, interpreting `x` by an infinite sequence $(x_i)_{i \in \mathbb{N}}$ is, for all $n \in \mathbb{N}$:

$$
\begin{array}{lcl}
v_n & = & x_n \\
memo_0 & = & 0 \qquad\qquad memo_{n+1} \;=\; \mathit{if}\; v_n \;\mathit{then}\; 0 \;\mathit{else}\; memo_n + memo_n \\
z_n & = & memo_n
\end{array}
$$

a. There was no question of concurrent programming in the work of Ashcroft and Wadge.

Lustre = LUCID Synchrone et Temps-réel [POPL'87]

Note the resenblance with what you have done in the course "Digital Systems"

```
node m(x: int) returns (z: int);
  var v: int; memo: int;
  let v = x;
      memo = 0 -> pre (if v then 0 else memo + memo);
      z = memo;
  tel;
```

LUCID was intended to be **general purpose**, an alternative to imperative lang.

— It provided a richer way to index values (not only integers).
— But semantics was not that of Kahn.
— This made the language difficult to compile efficiently and to be an alternative to imperative programming.

Lustre is a "synchronous" restriction of LUCID with a Kahn semantics.

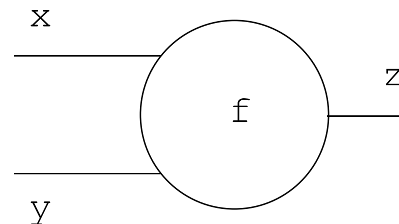
Come back to Kahn processes

We have essentially considered two kinds of nodes.

1. The point-wise application of a function.

```
let lift2 f x y z () =
  while true do
    let v = x.get () in
    let w = y.get () in
    z.put (f v w)
  done
```

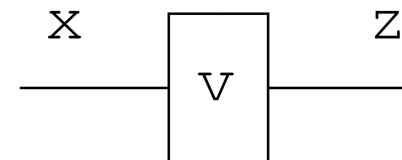
$z = f(x, y)$



2. Initialize a buffer *vs* emiting before reading. On can initialize a buffer with some value when creating it or simply reverse the order or read and write.

```
let unit_delay v x z =
  let memo = ref v in
  while true do
    z.put !memo;
    memo := x.get ()
  done
```

$z = \texttt{delay}_v(x)$



Exercice Show the encoding of the previous example (`main`) using those operators only. Prove it equivalent.

Come back to Kahn processes : what about control ?

We have only considered a set of processes that produce one output every time they consume one input.

There are several other situations : consumming/producing according to a boolean value, consumming/producing a fix (periodic) number of values.

1. E.g., a deterministic merge :

```
let merge c x y z () =
  while true do
   let v = c.get () in
   let w = if v then x.get () else y.get () in
   z.put w
  end
```

2. E.g., filtering according to a boolean condition.

```
let when c x z () =
  while true do
   let v = x.get () in let cond = c.get () in
   if cond then z.put v
  done
```

Deterministic `merge` and `when` :

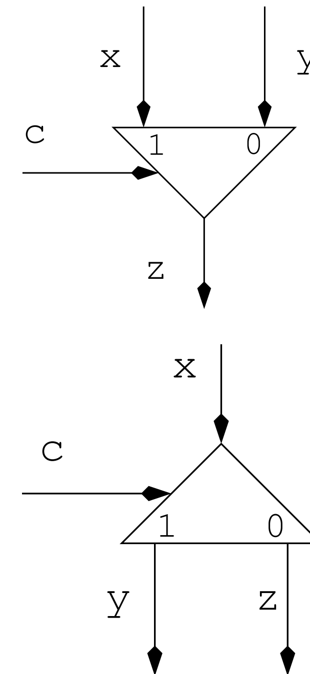
The consumption/production of data depend on values computed at run-time.

E.g., in the first process, every time `c` is read, `x` is read when `c` is true ; `y` is read when `c` is false. In the second, an output is produced when an input is consumed and evaluates to the value `true`.

Graphically :

$$z = \texttt{merge}\ c\ x\ y$$

$$y = x\ \texttt{when}\ c \quad \text{and} \quad z = x\ \texttt{whenot}\ c$$



Particular case : c is a periodic binary word (e.g., $1\,0\,1\,1\,0\,1\,1\,0\,1\ldots = (1\,0\,1)$).

Periodic consumption/production

The particular case of Kahn process networks where the number of read and write tokens is constant is called "Synchronous Data-flow" (or SDF). It was introduced by Lee & Messerschmitt [Proceedings of the IEEE, 1987].

1. E.g., Initialize a buffer with 3 values. Equivalently, first produce 3 values then act as the identity function.

```
let init x z () =
  z.put 0; z.put 1; z.put 2;
  while true do z.put (x.get ()) done
```

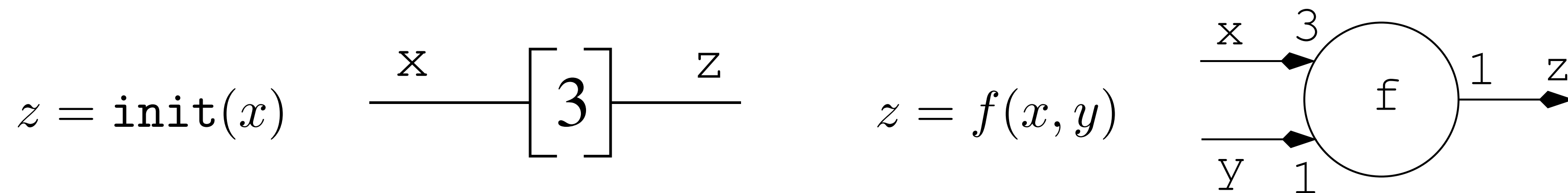
2. E.g., every time f make a step, it gets 3 values on x, 1 on y and puts 1 value on z.

```
let f x y z () =
       while true do
        let l = [|x.get (); x.get (); x.get ()|] in
        let result = scalar_product l [|0.2; 0.6; 0.2|] in
        z.put (if y.get () then 0.0 else result)
       done
```

This is represented by marking the transition and puting weights on channels.

$z = \texttt{init}(x)$

$z = f(x, y)$



An SDF is : A set of numbered nodes $S = \{N_1, ..., N_m\}$ with an output label $out : S \times S \rightarrow I\!N$, an input label $in : S \times S \rightarrow I\!N$, a delay function $delay : S \times S \rightarrow I\!N$.

$out(n_1, n_2)$ is the weight from n_1 to n_2. $in(n_1, n_2)$, the input weight of n_2 from n_1.

Static scheduling of SDF : A schedule $\sigma : I\!N \mapsto 2^S$. σ is periodic with period $p \in I\!N$ when forall $i \in I\!N$, $\sigma(p \times i) = \sigma(i)$. If an SDF is schedulable, than the set of "balanced equations" is solvable (for non trivial values of k_{n_i}) :

$$\wedge_{i,j} \; (k_{n_i} \times in(n_j, n_i)) = (k_{n_i} \times out(n_i, n_l))$$

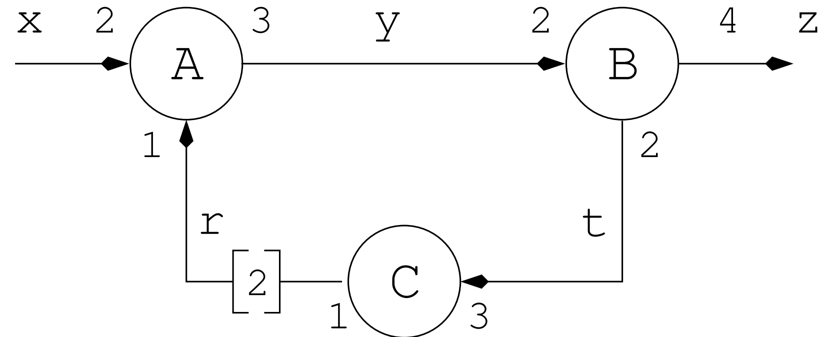
where $k_{n_i} \in I\!N$ means that node n_i is executed k_{n_i} times. That is, find $k = (k_{n_1}, ..., k_{n_m})^T \neq 0^T$ such that $(In - Out)\, k^T = 0^T$ where $In_{i,j} = in(n_i, n_j)$ and $Out_{i,j} = out(n_i, n_j)$.

Moreover, every feedback loop must be cut be an initialized delay.

Scheduling heuristics : The simplest is ASAP (As soon as possible) heuristic with a list scheduling algorithm.

Various optimization criteria (minimise total buffer size, minimize latency, i.e., time to get the first output).

Example of an SDF :



Balanced equations : $3 \times k_A = 2 \times k_B \quad 2 \times k_B = 3 \times k_C \quad k_A = k_C$

On average, execute $k_A = 2$ times A, $k_B = 3$ times B and $k_C = 2$ times C. Several schedules are possible : e.g., $(AA\,BBB\,CC)$, $(ABAB\,C\,BC)$, etc.

Note that replacing 1 on input of A by 2 leads to a non schedulable graph.

A lot of extensions of SDF : There is a wide range of SDF-like formalisms. Schedulability, deadlock analysis are well-known, for details). E.g., :

- "Marked graphs" are SDF where all weight equal 1. Dedicated algorithms.
- Extensions like CSDF ("cyclo-static SDF) with weight defined by periodic words (e.g., $(1\,4\,2)$), Boolean Data-flow (BDF), etc.
- Several prog. languages are based on the SDF : StreamIt (MIT), etc.

Yet, SDF-like formalisms are of limited expressiveness :

— They deal well with periodic behavior. An aperiodic signal (e.g, an emergency) would have to be sustained.
— The mix of periodic and non periodic becomes ad-hoc with no unified theory of the whole.
— It does not express precise timing constraints (e.g., the previous example could be scheduled in two different ways with radically different timing behaviors).
— They do not express dynamic reconfiguration in a modular manner.

What is a “reconfiguration” ?

E.g., press the cancel (reset) button on a phone, a boiler, an ATM. E.g., kill a process on reception of a signal then restart it (or a new one).

You have already experienced this feature : e.g., a Unix process which executes the `exec` system call.

This is a key feature but more imperative in style. Yet, it can also be given a functional encoding.

Dynamic reconfiguration

Let $f : V^\infty \mapsto V^\infty$. Define $\mathit{reset}\, f(x)\, \mathit{every}\, c$ for the function which acts like $f(x)$ and restarts every time c is true.

$$
\begin{aligned}
\mathit{reset}\, f(x)\, \mathit{every}\, c &= \epsilon \text{ if } x = \epsilon \text{ or } c = \epsilon \\
\mathit{reset}\, f(x)\, \mathit{every}\, c &= \mathit{whilenot}\, c\, \mathit{do}\, f(x)\, \mathit{then}\, (\mathit{reset}\, f(x\, \mathit{after}\, c)\, \mathit{every}\, (c\, \mathit{after}\, c))
\end{aligned}
$$

where :

$$
\begin{aligned}
\mathit{whilenot}\, c\, \mathit{do}\, x\, \mathit{then}\, y &= \mathit{whilenot}'(\texttt{false})(c)(x)(y) \\
x\, \mathit{after}\, c &= \mathit{after}'(\texttt{false})(x)(c) \\
\mathit{whilenot}'(v)(c)(x)(y) &= \epsilon \text{ if } c = \epsilon \text{ or } x = \epsilon \text{ or } y = \epsilon \\
\mathit{whilenot}'(v)(w.c)(y)(z) &= z \text{ if } v \wedge w \\
\mathit{whilenot}'(v)(w.c)(m.y)(z) &= m.\mathit{whilenot}'(\texttt{true})(c)(y)(z) \text{ otherwise} \\
\mathit{after}'(v)(x)(c) &= \epsilon \text{ if } x = \epsilon \text{ or } c = \epsilon \\
\mathit{after}'(v)(x)(m.c) &= x \text{ if } v \wedge m \\
\mathit{after}'(v)(w.x)(m.c) &= \mathit{after}'(\texttt{true})(x)(c) \text{ otherwise}
\end{aligned}
$$

Functions $\mathit{reset}\, .(.)\, \mathit{every}\, .$, $\mathit{whilenot}\, .\, \mathit{do}\, .\, \mathit{then}\, .$ and $.\, \mathit{after}\, .$ are also continuous.

Note that *reset* .(.) *every* . is a recursive function with a form very similar to "tail recursion" in classical (scalar) functional programming.

In the sequel, we shall consider a general case of data-flow network based on a the following basic operators :

- Lifting a scalar constant into a constant ; stream ;
- lifting a function to apply pointwise to its argument ;
- a unit initialized delay ;
- filtering (sampling), using `merge` and `when` ;
- a reset.

Programs are in SSA (Static Single Assignment) form.

Exercice (difficult) : Build a translation from an simple imperative language to a set of data-flow equations. Each imperative variable is interpreted as an history (a sequence).

The data-flow model of programs

The data-flow denotational model of a program can be used for several purposes :

— Program analysis and verification : write an imperative program (e.g., C + openMP libraries) with parallel composition through FIFOs. Build the data-flow representation and prove properties on it : deadlock analysis, buffer sizes, static scheduling, etc.

  A lot of new research here to analyze programs written in C, Java, etc.

— This can even be done on a sequential program in order to build its data-flow representation. Transforms it, apply optimizations on it then rebuild a "better" sequential implementation. Open question : what would be the extension of SSA representation used in opt. compilers (e.g., GCC, LLVM) to deal with time and low level concurrency ?

  A lot of new research here too. Links with **Polyhedral compilation**.

— Directly write data-flow equations and go the opposite way : build an implementation, sequential, parallel or a combination of both.

  Essentially what **synchronous data-flow languages** have done.

Interest/weaknesses of the Kahn semantics

The good :

- **Functional :** simple semantics, parallelism **and** determinism. The semantics does not depend on the actual scheduling of activities.
- **Modularity, compositionality :** the fix-point of a set of equations still defines a continuous function.
- **Time insensitive :** slowing down inputs does dot modify the system behavior.
- **GALS (Globally Asynchronous, Locally Synchronous) :** Every process is a sequential program while the set run asynchronously.

The bad :

- **Time insensitive :** it is impossible to predict the computation time, to interrupt on a time-out.
- **No guaranty on the memory :** buffer sizes may be unbounded (no meaning in hardware, not satisfactory for software).
- **No test of absence** it is impossible to stop watching on a buffer depending on a timer.
- **Inefficient :** scheduling is dynamic.

Synchronous Kahn Networks

Keep the Kahn semantics but restrict to networks that can be **statically scheduled** and communication reduces to a simpler mechanics with **statically bounded buffers**. Provides **clocks as types** to express synchronization constraints.

Examples :

1. Synchronous circuits : a restricted case of a Kahn process network where communication between processes is instantaneous (zero buffer).
2. Synchronous data-flow languages such as Lustre.
3. SDF (Synchronous Data-flow), N-synchronous systems.
4. Kahn networks with bounded buffers : model bounded buffers as processes with back-pressure edges.

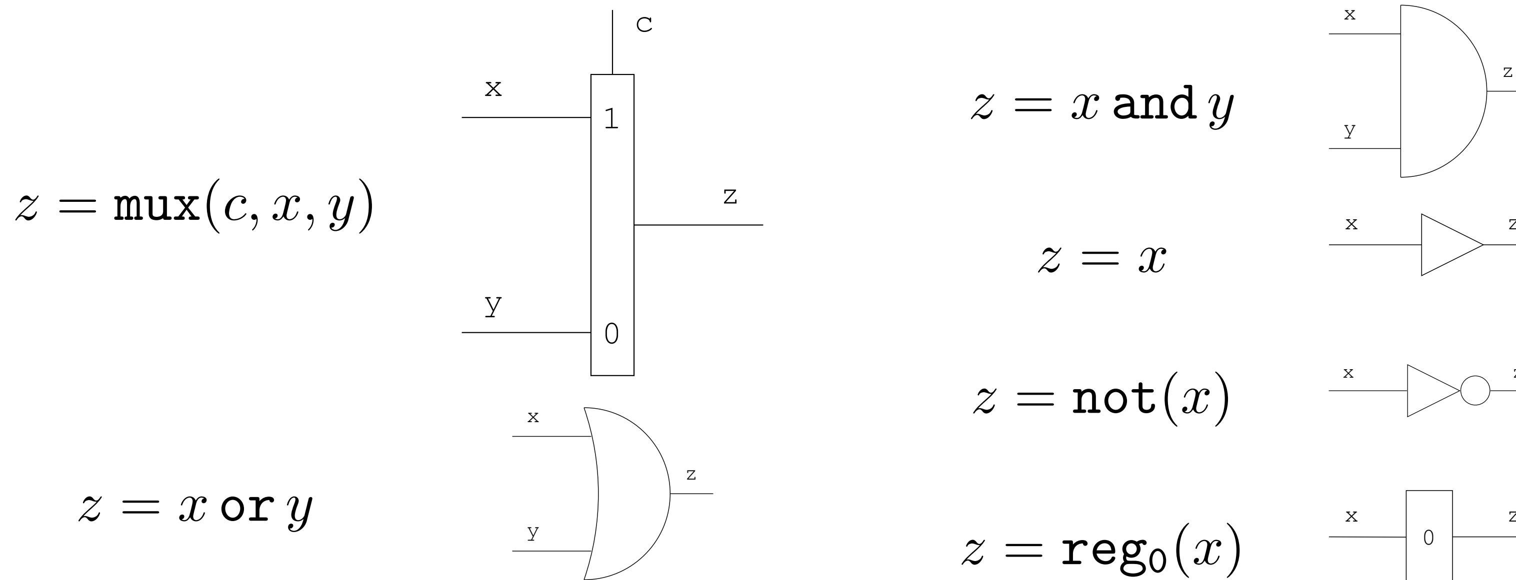
Example : (single clocked) synchronous circuits

— Boolean values are noted 0 (false) and 1 (true).

— A set of boolean operators : **or**, **and**, **mux**, **not**.

— A unit delay $\mathtt{reg}_0$ (or register) initialized to false (0).

Graphical representation :



Semantics :

— $V^\infty = V^* + V^\omega$, set of finite and infinite sequences of elements in V.

— The empty sequence ϵ. $x.y$ whose head is x, rest is y.

Operators as stream functions

$$
\begin{array}{lcll}
\texttt{mux}(c, x, y) & = & \epsilon & \text{if } x = \epsilon \text{ or } y = \epsilon \text{ or } c = \epsilon \\
\texttt{mux}(0.c, v.x, w.y) & = & w.\texttt{mux}(c, x, y) & \\
\texttt{mux}(1.c, v.x, w.y) & = & v.\texttt{mux}(c, x, y) & \\
x \,\texttt{or}\, y & = & \epsilon & \text{if } x = \epsilon \text{ or } y = \epsilon \\
v.x \,\texttt{or}\, w.y & = & (v \vee w).x \,\texttt{or}\, y & \\
x \,\texttt{and}\, y & = & \epsilon & \text{if } x = \epsilon \text{ or } y = \epsilon \\
v.x \,\texttt{and}\, w.y & = & (v \wedge w).x \,\texttt{or}\, y & \\
\texttt{not}(\epsilon) & = & \epsilon & \\
\texttt{not}(v.x) & = & \neg v.\texttt{not}(x) & \\
\texttt{reg}_0(x) & = & 0.x &
\end{array}
$$

Remark : More generaly, delays and point-wise application are :

$$
\begin{array}{lcl}
\texttt{delay}_v(x) & = & v.x \\
\texttt{lift2}\, f\, x\, y & = & \epsilon \text{ if } x = \epsilon \text{ or } y = \epsilon \\
\texttt{lift2}\, f\, (v.x)\, (w.y) & = & (f\, v\, w).\texttt{lift2}\, f\, x\, y
\end{array}
$$

Example : what is computed by the equation $x = \texttt{not}(\texttt{reg}_0(x))$?

Let us try to compute it iteratively, starting with ϵ. ϵ means "I don't know the value of x". Let $x^0 = \epsilon$. We compute the sequence $x^n = \texttt{not}(\texttt{reg}_0(x^{n-1}))$:

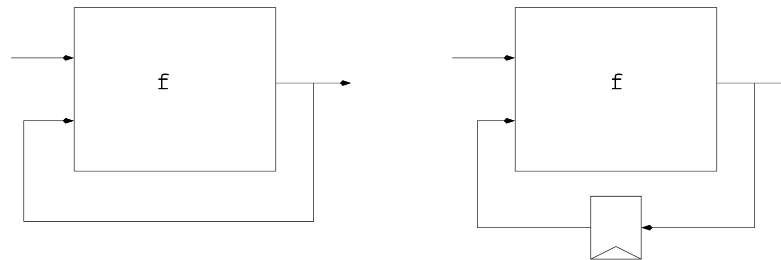
$$
\begin{array}{lllll}
x^0 & = & \epsilon & & \\
x^1 & = & \texttt{not}(\texttt{reg}_0(x^0)) & = & \texttt{not}(0.\epsilon) & = & 1.\epsilon \\
x^2 & = & \texttt{not}(\texttt{reg}_0(x^1)) & = & \texttt{not}(0.1.\epsilon) & = & 1.0.\epsilon \\
\ldots & & & & \\
x^n & = & x_1....x_n & \text{with} & x_i = \neg x_{i-1}
\end{array}
$$

The limits of this iteration defines the infinite sequence $x = x_0.x_1...$ such that $\forall i > 1, x_i = \neg x_{i-1}$ and $x_0 = 0$.

This iterative process defines *more and more* the solution of the equation $x = \texttt{not}(\texttt{reg}_0(x))$. Because all operators are continuous on $(\{0+1\}^\infty, \leq, \epsilon)$, this is the smallest solution.

Causality (feedback loops)

What happens when the output of a block is connected to one of its inputs ?



- the smallest solution of equations $X = X$ or $X = X\,\texttt{or}\,Y$ is ϵ
- the smallest solution of $X = \texttt{not}(\texttt{reg}_0(X))$ is $X = 1.0.1.0....$

Checking causality : Which static condition should we take to ensure that the solution is not ϵ ?

This is (in general) a non decidable problem. Most tools based on block-diagram formalisms (e.g., **Scade**, **Simulink**) reduce it to "syntactical" causality : every feedback loop must cross a delay (register)

Sufficient condition for causality

A sufficient condition for causality is that every feed-back loop crosses an explicit delay (e.g., `reg`). We call it "syntactic causality", as it does not depend on dynamic, boolean condition.

Property 1 (Syntactic causality) *Let $f : V^\infty \mapsto V^\infty$ be a continuous, length preserving function. If $\epsilon < f(\epsilon)$ then $fix\ f \neq \epsilon$*

— a delay (initialized to v) : $\texttt{reg}_v(X) = v.X$
— i.e., causality is essentially based on a length argument.

Note that the property $\epsilon \neq f(\epsilon)$ means that f is non-strict.

Strictness analysis : (lazy-functional languages [**?**]) a function f is strict if it does need its argument in order to produce its output, i.e., $f(\bot) = \bot$.

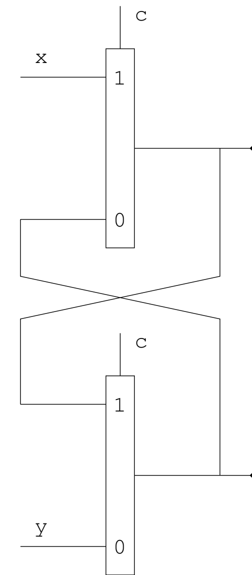
Causality is related to a very old property called "productivity" or "liveness" in list/stream programs : under which condition a fix-point does not stuck. E.g., read [**?**, **?**] in the case of typed functional languages.

Constructiveness causality

Some networks have cycles (or *instantaneous loops*) that do not cross delay. Yet, they are perfectly causal : feeding inputs with a infinite sequence of values produce a unique infinite sequence of values.

E.g., take a system of equations with inputs c, x, y and produce z, t.

$$f(c, x, y) = (z, t) \textit{ where } \begin{aligned} z &= \texttt{mux}(c, x, t) \\ t &= \texttt{mux}(c, z, y) \end{aligned}$$



f is equivalent to $g(c, x, y) = \texttt{mux}(c, x, y), \texttt{mux}(c, x, y)$. Causality and constructivity has been studied by Berry et al. for the language Esterel. Constructiveness corresponds to proving a property in intuitionistic logic (read [the constructive logic of Esterel, Berry, 1989]).

Implementation into Kahn networks

When `c` is true, one must only read the other input after having produced the output (and conversely).

```
let mux c x y o () =
  while true do
    let v = c.get () in
    if v then let w = x.get () in
              o.put w;
              ignore (y.get ())
    else let w = y.get () in
         o.put w;
         ignore (x.get ())
  done
```

Constructivness of Esterel is more expressive : a circuit is constructive if it stabilizes and this take the semantics of boolean operators (`or`, `&`, `not`) into account (not only `mux`). E.g., if $x = 1$ then x `or` $y = 1$ (the same for $y = 1$).

Question : Is-this extra expressiveness necessary for the compilation of Esterel into circuits ?

Lustre

In a first approximation, we have defined a subset of Lustre. To get a more realistic one, add other operators (e.g., on integers, etc.).

Moreover, we need a way to mix slow and fast processes, i.e., activate a computation, reading/producing a value according to a boolean condition.

Sampling : mixing slow and fast processes In Lustre and Lucid Synchrone, this is based on sub-sampling/over-sampling.

- `when` is a sampler which allows to go from a fast process to a slow one ;
- `merge` builds the union of two sequences.

$$
\begin{array}{lcl}
x \ \texttt{when} \ c & = & \epsilon \text{ if } x = \epsilon \text{ or } c = \epsilon \\
(v.x) \ \texttt{when} \ (\texttt{true}.c) & = & v.(x \ \texttt{when} \ c) \\
(v.x) \ \texttt{when} \ (\texttt{false}.c) & = & x \ \texttt{when} \ c \\
\texttt{merge} \ c \ x \ y & = & \epsilon \text{ if } c = \epsilon \text{ or } x = \epsilon \text{ or } y = \epsilon \\
\texttt{merge} \ (\texttt{true}.c) \ (v.x) \ y & = & v.\texttt{merge} \ c \ x \ y \\
\texttt{merge} \ (\texttt{false}.c) \ x \ (v.y) & = & v.\texttt{merge} \ c \ x \ y
\end{array}
$$

These operators are continuous.

Implementation of streams

Let's program now with all these data-flow operators !

An element of $V^\infty = V^* \cup V^\omega$ can be represented in several ways :

1. A lazy data-structure $\mathit{Stream}(A)$ with a single constructor :

$$. : A \times \mathit{Stream}(A) \rightarrow \mathit{Stream}(A)$$

   The empty sequence ϵ is simply $\mathit{fail}()$, the program that never stop !

$$\mathit{fail}() = \mathit{fail}()$$

2. A prefix-closed function from $\mathbb{N}$ to V, that is, if $f(t)$ is defined, it is also the case for all $t' < t$. ϵ is a function with empty domain.

Both representations can be encoded (embedded) into an existing laby functional language. We try Haskell.

An Embedding in Haskell

A Tiny Lustre in Haskell

```
module Streams where

data ST a = Cons a (ST a) deriving Show

-- lifting constants
constant x = Cons x (constant x)

-- pointwise application
extend (Cons f fs) (Cons x xs) = Cons (f x) (extend fs xs)

-- delays
(Cons x xs) `fby` y = Cons x y
pre x y = Cons x y
```

That's all : we get recursion, function composition, higher-order, type inference for free.

Remark : the same kind of encoding can be done in Ocaml (using module `Lazy`, see notes).

Define basic operators, e.g. :

```
lift0 x = constant x
lift2 f x y = extend (extend (constant f) x) y
lift3 f x y z = extend (extend (extend (constant f) x) y) z

x `plusl` y = lift2 (+) x y
x `minusl` y = lift2 (-) x y
x `divl` y = lift2 (div) x y
x `eql` y = lift2 (=) x y

zero = lift0 0
one = lift0 1
zerof = lift0 0.0
always = lift0 true
never = lift0 false

ifthenelse x y z = lift3 (\x,y,z -> if x then y else z) x y z
x `init` y = ifthenelse (always `fby` never) x y -- x -> y in Lustre
pre0 x = pre 0 x
reg0 x = pre false x
```

Examples

```
-- integers greaters than n
from n = let nat = n `fby` (nat `plusl` one) in nat

-- resetable counter
reset_counter res input =
  let output = ifthenelse res zero v
      v = ifthenelse input (zero `fby` (output `plusl` one))
                           (zero `fby` output)
  in output

-- a periodic boolean sequence 1 0 0 0 ... 1 0 0 0 ...
every n =
  let cpt = reset_counter o always
      o = always `init` ((pre0 cpt) `eql` (n `minusl` one)) in
  o
```

Exercice : Programs can be writen in a more natural, say Lustre syntax. Write a parser so that `1 + 2` is interpreted as ``(constant 1) `plusl` (constant 2)``.

The previous examples are written this way, in Lustre :

```
node from(n: int) returns (nat: int);
  let nat = n -> pre (nat + 1); tel;

node reset_counter(res, input: bool) returns (output: int);
  let
    output = if res then 0 else v;
    v = if input then 0 -> pre (output + 1) else 0 -> pre output;
  tel;

node every(n: int) returns (o: bool);
  var cpt: int;
  let
    cpt = reset_counter(o, true);
    o = true -> pre cpt = (n - 1);
  tel;
```

Sampling (when and merge)

```
module Streams where
...
-- sampling
(Cons x xs) `when` (Cons True cs)  = Cons x (xs `when` cs)
(Cons x xs) `when` (Cons False cs) = xs `when` cs

merge (Cons True c)  (Cons x xs) y  = Cons x (merge c xs y)
merge (Cons False c) x (Cons y ys)  = Cons y (merge c x ys)

x `whenot` c = x `when` (lift1 not c)
```

Example : model periodic sampling

1. If $(top_k)_{k \in \mathbb{N}}$ is a sequence, compute the sub-sequence $(top_{n.k})_{k \in \mathbb{N}}$. Simply define the function :

```
filter k top = top when (every k)
```

2. Define a clockwatch that produces seconds, minutes and hours. `second` is a sub-sequence of `mili` ; `minute`, a sub-sequence of second, etc.

```
hour_minute_second mili =
  let second = filter (constant 1000) mili in
  let minute = filter (constant 60) second in
  let hour = filter (constant 60) minute in
  hour, minute, second
```

Activation condition

A classical operation in block diagrams is the "activation condition". It activates a block on a boolean condition. [a]

E.g., consider a PID controller. Given an input e, produce an output pid by adding a response proportionally to e, the integration of e and the derivative of e.

$$pid(e)(t) = p \times e(t) + i \times \int_0^t e(\tau)d\tau + d \times \dot{e}(t)$$

Now, replace the integration by a fix-step forward Euler method and the derivative by a difference. Take h, as the integration step. For all $n \in \mathbb{N}$, suppose that the sequence $e'(n)$ approximates $e(n \times h)$. Then, $pid'(e)(n)$ approximates $pid(e)(n \times h)$.

$$pid'(e')(n) = p \times e'(n) + i \times \sum_0^n h \times e'(n) + d \times (e'(n) - e'(n-1))/h$$

What to do if e' arrives every one milisecond ? We have to compute $pid'(e')$ every h

a. See "activation condition" in **Scade** ; "enabled subsystem" and "enabled and triggered subsystem" in **Simulink**.

milisecond only. This is done by sampling. Defines :

$$
\begin{array}{rcl}
pid''(e')(n) & = & pid'(e')(n) \text{ if } n \, mod \, h = 0 \\
pid''(e')(n) & = & pid''(e')(n-1) \text{ otherwise}
\end{array}
\quad \text{i.e.,} \quad pid''(e')(n) \quad = \quad pid'(e')(\lfloor n/h \rfloor)
$$

Written in Lustre :

```
node pid'(const p, i, d, h: real; e':real) returns (o: real);
  var proportional, integral, derivative: real;
  let
     proportional = p * e';
     integral = 0.0 -> pre (integral + h * e');
     derivative = 0.0 -> (e' - pre e) / h;
     o = proportional + integral + derivative;
  tel;

node pid''(const p, i, d, h: real; e': real) returns (o: real);
  let o = current (pid'(p, i, d, h, e' when (every h))); tel;
```

E.g., take $h = 4$ miliseconds.

`e'`	=	e'_0	e'_1	e'_2	e'_3	e'_4	e'_5	e'_6	e'_7	e'_8	...
`one_over_h`	=	1	0	0	0	1	0	0	0	1	...
`e' when one_over_h`	=	e'_0				e'_4				e'_8	...
`pid'(e' when one_over_h)`	=	o_0				o_1				e_2	...
`pid''(e')`	=	o_0	o_0	o_0	o_0	o_1	o_1	o_1	o_1	o_2	...

We have built sequences, sub-sequences and have composed them.

`current` *versus* `merge` : The `current` operator of Lustre cannot be defined within our kernel language : "if x is a sequence, `current`(x) is a sequence faster than x". What does it mean to be "faster" ? How much ?

The operator `merge` : go from slow to fast processes.

`c`	=	1	0	0	1	1	0	0	1	0	...
`x`	=	x_0			x_1	x_2			x_3		...
`y`	=		y_0	y_1			y_3	y_4		y_5	...
`merge c x y`	=	x_0	y_0	y_1	x_1	x_2	y_3	y_4	x_3	y_5	...

In Haskell :

```
pid' p i d e' =
  let proportional = p * x in
  let integral = pre 0.0 (integral `plusl` (h `mull` e')) in
  let derivative =
        (constant 0.0) `init` ((x `minusl` (pre 0.0 e')) `divl` h) in
  proportional `plusl` (integral `plusl` derivative)

pid'' p i d h e' =
  let one_over_h = every h in
  let o = merge one_over_h (pid' p i d (e' `when` one_over_h))
                  ((pre 0.0 o) `whenot` one_over_h) in
  o

-- a generic (higher-order) activation condition

activation_condition c default f x =
  let o = merge c (f(x `when` c)) ((default `fby` o) `whenot` c) in o

pid'' p i d h e' =
  activation_condition (every h) (constant 0.0) (pid'' p i d h) e'
```

Bounded Oversampling

Over-sampling : the internal rate of a process is faster than the rate of inputs.

E.g., compute the stuttering sequence $(o_n)_{n \in I\!N}$ such that :

$$o_n = x_{n/k} \text{ if } n \ mod \ k = 0 \qquad o_n = o_{n-1} \text{ otherwise}$$

```
one_over_10 = every (constant 10)
stutter x =
  let o = merge one_over_10 x ((pre0 o) ‘whenot‘ one_over_10) in o
```

This program mimics an internal `for` loop of a process that reads a value in an input channel `x` and writes ten values in an output channel `o`. This can be seen as the denotation of the following input/output process :

```
let stutter x o () =
  while true do
    let v = x.get () in
    for i = 1 to 10 do
      o.put v
    done
  done
```

(possibly unbounded) oversampling

Take an input sequence $i = i_0.i_1....$ and computes an approximated value of $\sqrt{i} = \sqrt{i_0}.\sqrt{i_1}...$ using the Newton method.

An imperative implementation :

```
let eps = 0.001
let roots x o =
  while true do
    let v = x.get () in
    let pv = ref 0.0 in
    while abs (v -. pv) > eps do
      pv := (!pv /. 2.0) + v /. 2.0 *. !pv
    done;
    o.put !pv
  done
```

What is the semantics of this program, as a transformation of an input sequence $(i_k)_{k \in I\!N}$ into an output sequence $(o_k)_{k \in I\!N}$?

The internal loop computes the sequence :

$$u_1 = x \qquad u_n = u_{n-1}/2 + x/2u_{n-1}$$

```
eps = constant 0.001

-- zero holder
hold c default x =
  let o = merge c x ((default `fby` o) `whenot` c) in o

-- the root function
root eps x =
  let xc = hold ask (constant 0.0) x
      pu = ifthenelse ask zero) (pre0 u)
      u = (pu `divl` (constant 2.0))
          `plusl` (xc `divl` ((constant 2.0) `mul` pu))
      ask = always `fby` ok
      ok = absl ((u `minusl` pu)) `less` eps
      o = u `when` ok in
  o
```

E.g. :

`i` =	i_0			i_1				i_2		...
`ic` =	i_0	i_0	i_0	i_1	i_1	i_1	i_1	i_2		...
`uc` =	uc_0	uc_1	uc_2	uc_3	uc_4	uc_5	uc_6	uc_7	uc_8	...
`ok` =	0	0	1	0	0	0	1	0	0	...
`o` =			uc_2				uc_6			...

We have embedded in Haskell the semantics of a set of data-flow primitives, close to those of Lustre and Lucid Synchrone.

Good : we do not need any new programming language, type system, compiler, etc.

A DSL in Haskell

This idea of defining a language by a collection of operators whose semantics is defined in Haskell has been named “embedding Domain Specific Languages (DSL)”. It has been popularized in the late 90’s by Conal Elliot et al. with the DSL FRAN [Elliot et al., ICFP’97] and later on with Functional Reactive Programming (FRP) [Hudak et al., PLDI’00].

E.g., see Hawk [Launchburry et al., ICFP’99] for modeling processor architectures. Essentially based on the first three primitives (lifting constants, pointwise application, unit delay).

A DSL becomes a Haskell library. One can go a little further by giving different implementations to primitives. This idea has been first popularized by Sheeran et al. with the DSL Lava, a Hardware Description Language in Haskell [Sheeran et al., ICFP’98].

The Haskell class type system simplifies this. Several implementations of the primitives may give, e.g., (1) an interpretor ; (2) the generation of a net-list (circuit) and macro-generation of code ; (3) a test sequence, etc.

Efficiency

Yet, the Haskell implementation with signals represented as lazy streams is quite efficient.

E.g., compute the first 4 values of `x = from 1`. `x` is computed lazilly :

1. `x = nat` with `nat = Cons 1 nat1` with `nat1 = plusl nat (constant 1)`
2. `nat1 = plusl (Cons 1 nat1) (constant 1) = Cons 2 nat2`
   with `nat2 = plusl nat1 (constant 1)`
3. `nat2 = Cons 2 nat3` with `nat3 = plusl nat2 (constant 1)`
4. ...

The data-structure `nat` is built progressively, i.e., the first value is produced, then the second from the first one, then the third from the second one.

Signals as functions from integers to values

Note that this is far more efficient than an encoding of streams as functions from integers to values, e.g., :

```
module Streams where
type ST a = Int -> a

constant x n = x

extend f x n = (f n) (x n)

pre x y n = if n = 0 then x else y(n-1)

fby x y n = if n = 0 then x(0) else y(n-1)

when x y n = x(index y n)

merge c x y n = if c(n) then x(n) else y(n)

index x n = if x(n) then n else index x (n+1)
```

E.g., nothing is reused from step to step when computing the first four values of `x = from 1`. The computation of `x(`n`)` takes n steps.

This semantics is nonetheless simpler, in particular in a system which forbids unbounded recursion (e.g., Coq) and for program verification (e.g., modeling Lustre in a SMT solver).

The representation of sequences as a lazy data-structure does caching (store previous results) automatically.

For the computation of `x = from 1`, the overall memory is bounded as every step allocates a new `Cons` which becomes useless and is deallocated by the GC later on. Still, the execution is efficient but not that much :

1. It allocate/deallocates all the time and thus needs a GC ;
2. the run-time system of Haskell is huge ;
3. execution time is unpredictable ;
4. some programs are monsters and must be statically rejected.

Some programs generate monsters

A stream is represented as a lazy data-structure. Nonetheless, lazyness allows streams to be build in a strange manner. Indeed, in Haskell, the order between structures is that of Scott :

Structural (Scott) order :

$\perp \leq_{struct} v$, $(v : w) \leq_{struct} (v' : w')$ iff $v \leq_{struct} v'$ and $w \leq_{struct} w'$.

The following programs are perfectly correct in Haskell (with a unique non bottom solution)

```
first (x:y) = x
next (x:y) = y
incr (x:y) = (x+1) : incr y


one = 1 : one
x = (if hd(tl(tl(tl(x)))) = 5 then 3 else 4) : 1 : 2 : 3 : one
output = (hd(tl(tl(tl(x))))) : (hd(tl(tl(x)))) : (hd(x)) : output
```

Their value is :

— $x = 4 : 1 : 2 : 3 : 1 : ...$

— $output = 3 : 2 : 4 : 3 : 2 : 4 : ...$

These streams may be constructed lazilly :

— $x^0 = \bot, x^1 = \bot : 1 : 2 : 3 : un, x^2 = 4 : 1 : 2 : 3 : one.$

— $output^0 = \bot, output^1 = 3 : 2 : 4 : ...$

An other example (due to Paul Caspi) :

```
nat = zero ‘fby‘ (incr nat)
ifn n x y = if n <= 9 then hd(x) : if9(n+1) (tl(x)) (tl(y)) else y
if9 x y = ifn 9 x y


x = if9 (incr (next x)) nat
```

We have $x = 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 10, 11, ...$

Are they reasonnable programs ? Streams are constructed in a reverse manner from the future to the past and are not “causal”.

This is because the structural order between streams allows to fill the holes in any order, e.g. :

$$(\bot : \bot) \leq (\bot : \bot : \bot : \bot) \leq (\bot : \bot : 2 : \bot) \leq (\bot : 1 : 2 : \bot) \leq (0 : 1 : 2 : \bot)$$

It is also possible to build streams with intermediate holes (undefined values in the middle) through the final program is correct :

$$half = 0.\bot.0.\bot...$$

```
fail = fail
half = 0:fail:half
fill x = (hd(x)) : fill (tl(tl x))
ok = fill half
```

We shall need to model **causality**, that is, stream should be produced in a sequential order. We take the **prefix order**.

Prefix order :

$$x \leq y \text{ if } x \text{ is a prefix of } y \text{, that is : } \bot \leq x \text{ and } v.x \leq v.y \text{ if } x \leq y$$

Causal function :

Causal computations shall be modeled by considering monotone functions :

$$x \leq y \Rightarrow f(x) \leq f(y)$$

All the previous program will get the value $\bot$ in the Kahn semantics.

Kahn Semantics in Haskell

It is possible to remove possible non causal streams by forbidding values of the form $\bot.x$. In Haskell, the annotation `!a` states that the value with type `a` is strict ($\neq \bot$). See file `SStreams.hs` :

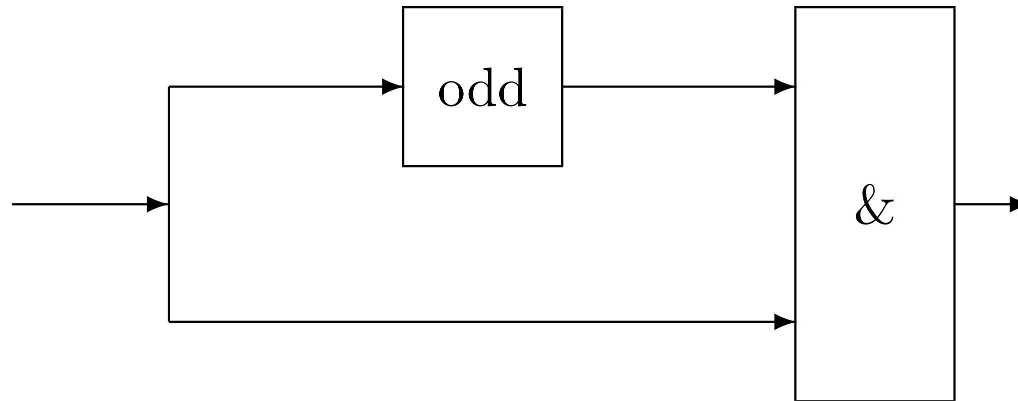
```
module SStreams where
-- only consider streams where the head is always a value (not bot)
data ST a = Cons !a (ST a) deriving Show
constant x = Cons x (constant x)

extend (Cons f fs) (Cons x xs) = Cons (f x) (extend fs xs)

(Cons x xs) `fby` y = Cons x y

(Cons x xs) `when` (Cons True cs)  = (Cons x (xs `when` cs))
(Cons x xs) `when` (Cons False cs) = xs `when` cs

merge (Cons True c)  (Cons x xs) y  = Cons x (merge c xs y)
merge (Cons False c) x (Cons y ys)  = Cons y (merge c x ys)
```

This time, all the previous non causal programs have value $\bot$ (stack overflow).

Exercice : provide the same definitions in Ocaml. Build a fix-point operator to program stream equations.

Some "synchrony" monsters

What appears when a stream is consummed on different rates ?




```
odd x = x `when` half
non_synchronous x = andl x (odd x)
```

Unbounded FIFOs (Heap overflow in Haskell)

Remarks :

- Each operator is finite-memory (a combinatorial `andl` gate + a half frequency sampler `when half`) through the composition is not.
- Everything is hidden in the communication channels. The Haskell encoding hides all the memory and synchronization mechanisms.

We need to fine-tune the semantics.

Conclusion of this encoding in Haskell

We could provide others (an more efficient) encoding. Yet, causality and synchronization issues would stay and have to be statically detected.

- The encoding does not model synchrony and causality issues, i.e., deadlocks and values arriving at the same time or not.
- We need static analysis to **reject** programs.
- There are subtle typing issues as every operator comes in twice versions. E.g., : `1`, `constant 1`, ,, `lift2 (,)`, etc. which calls for explicit conversions everywhere : `stream_of_pair`, `pair_of_stream`. Programming becomes a nightmare. [a]
- For embedded software, we need far better time and resource predictable code than what the Haskell run-time system provides : a compiler is needed to generate code that is proved to run in bounded time and memory.

None of this can be done for free by a simple embedding in Haskell ; this is the difficult part. A lot of research has been done arround FRP and DSL in Haskell, for almost 15 years. It has not been very convincing.

"A good language is a language that reject programs !"

a. To convince yourself, program an application in Hawk.

Conclusion (for this lesson)

- The Kahn semantics gives a precise meaning of what is a set of deterministic process networks communication through FIFOs.
- We need to restrict the expressiveness of Kahn networks to avoid severe issues, causality, clock (synchronization) issues and boundedness of memory and time.
- Yet, we have only provided a denotational semantics. An operational one that explain compilation and how resources are used is lacking.
- Yet, statically analysis to ensure that programs can be executed synchronously is lacking.

A suivre...